# Concurrency в Go

# Как проходит занятие

- Активно участвуем - задаем вопросы.
- Чат вижу - могу ответить не сразу.
- После занятия - оффтопик, ответы на любые вопросы.

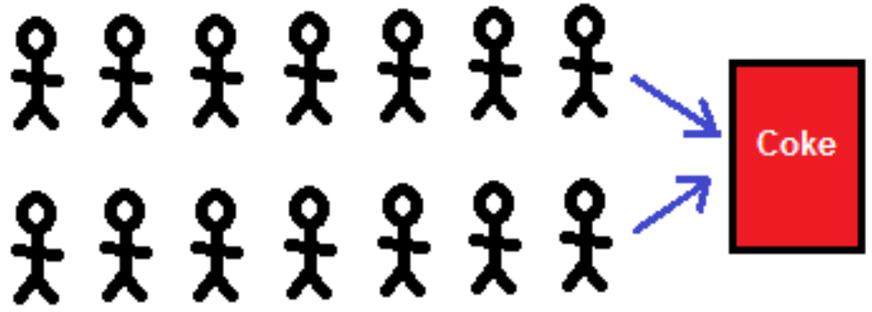
# О чем будем говорить

- Горутины
- Каналы
- WaitGroup
- Once
- Mutex

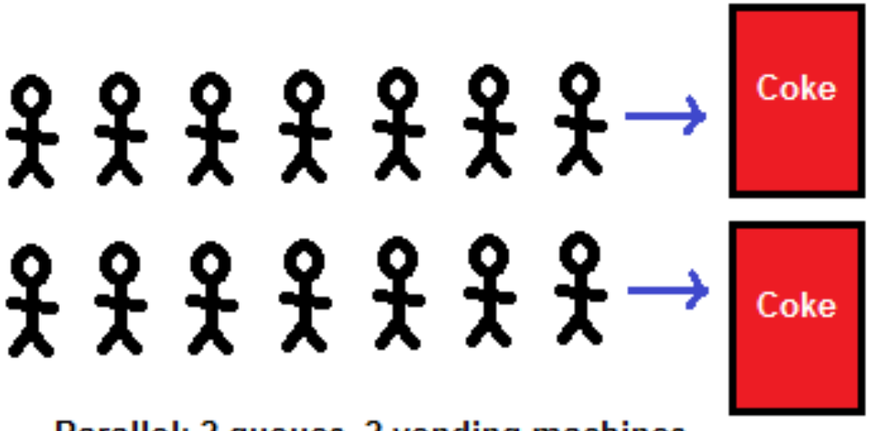
# Горутины

Горутины — функции, которые выполняются конкурентно. Горутины — легковесные, у каждой из них свой стек, все остальное (память, файлы и т.п.) — общее.

# Параллелизм vs конкурентность

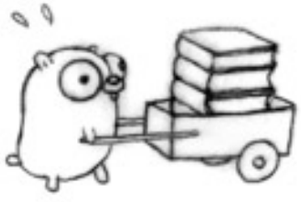



Concurrent: 2 queues, 1 vending machine



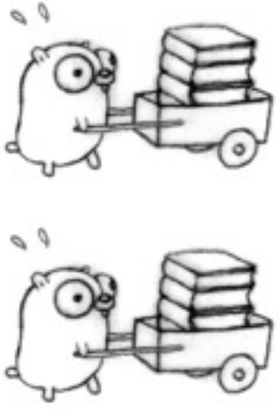

Parallel: 2 queues, 2 vending machines

# Параллелизм vs конкурентность

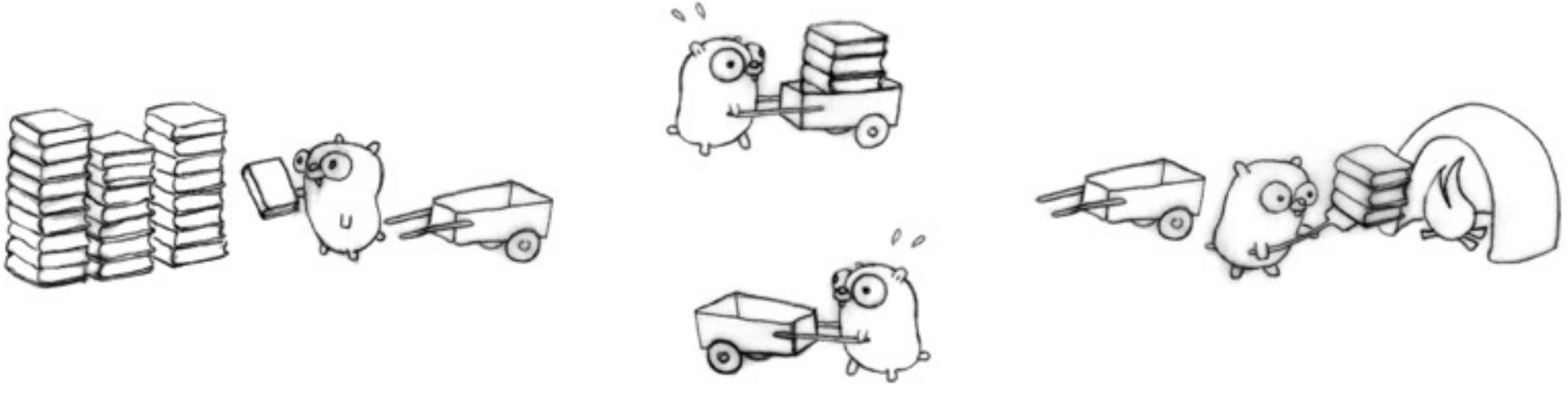


Картинки из выступления Роба Пайка: https://blog.golang.org/waza-talk

# Параллелизм vs конкурентность



# Параллелизм vs конкурентность



# Запуск горутины

```
go func(trolley Trolley) {
    burnBooks(trolley)
}(trolley)
```

```
go func() {
    burnBooks(trolley)
}()
```

```
go burnBooks(trolley)
```

```
go trolley.Load(pile)
```

# Как работают горутины?

- Когда запускается горутина?
- .
- .

# Как работают горутины?

- Когда запускается горутина?
- Когда горутина приостанавливается?
- .

# Как работают горутины?

- Когда запускается горутина?
- Когда горутина приостанавливается?
- Что будет с горутинами, если программа закончится?

# Сколько тут горутин?

```
func main() {
    fmt.Printf(
        "Goroutines: %d",
        runtime.NumGoroutine(),
    )
}
```

https://goplay.space/#8is1y5tu-m3

# Что напечатает программа?

```
func main() {
    go fmt.Printf("Hello")
}
```

https://goplay.space/#SBO2dnLQPue

# Замыкание

```
for i := 0; i < 5; i++ {
    go func() {
        fmt.Print(i)
    }()
}
time.Sleep(2 * time.Second)
```

https://goplay.space/#rSKy5YetcJS

# Каналы

```
chan T
```

- работают с конкретным типом
- потокобезопасны
- похожи на очереди FIFO

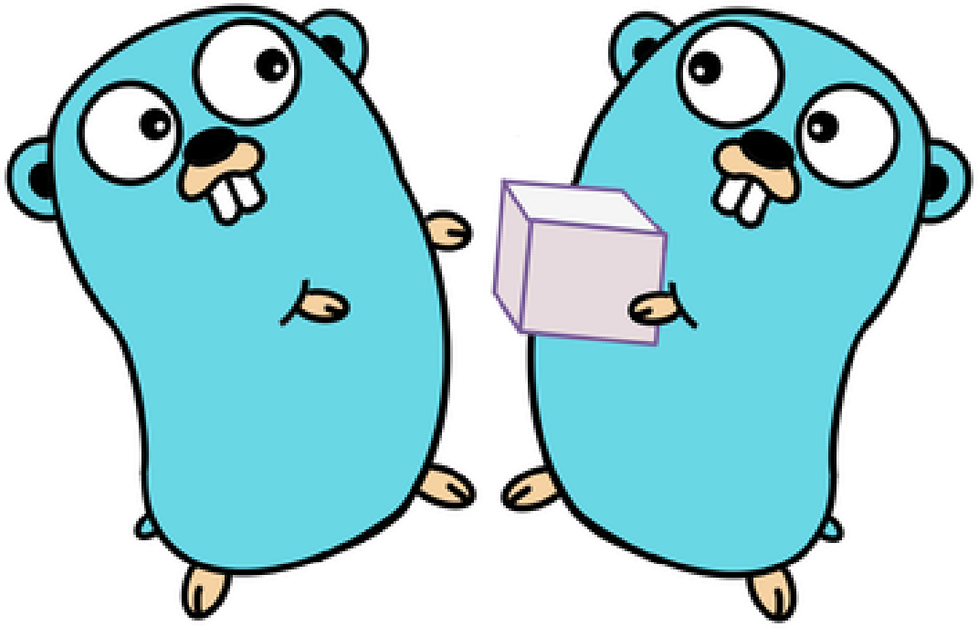
# Каналы: операции

```
ch := make(chan int) // создать канал
ch <- 10             // записать в канал
v := <-ch            // прочитать из канала
close(ch)            // закрыть канал
```

https://habr.com/ru/post/308070/

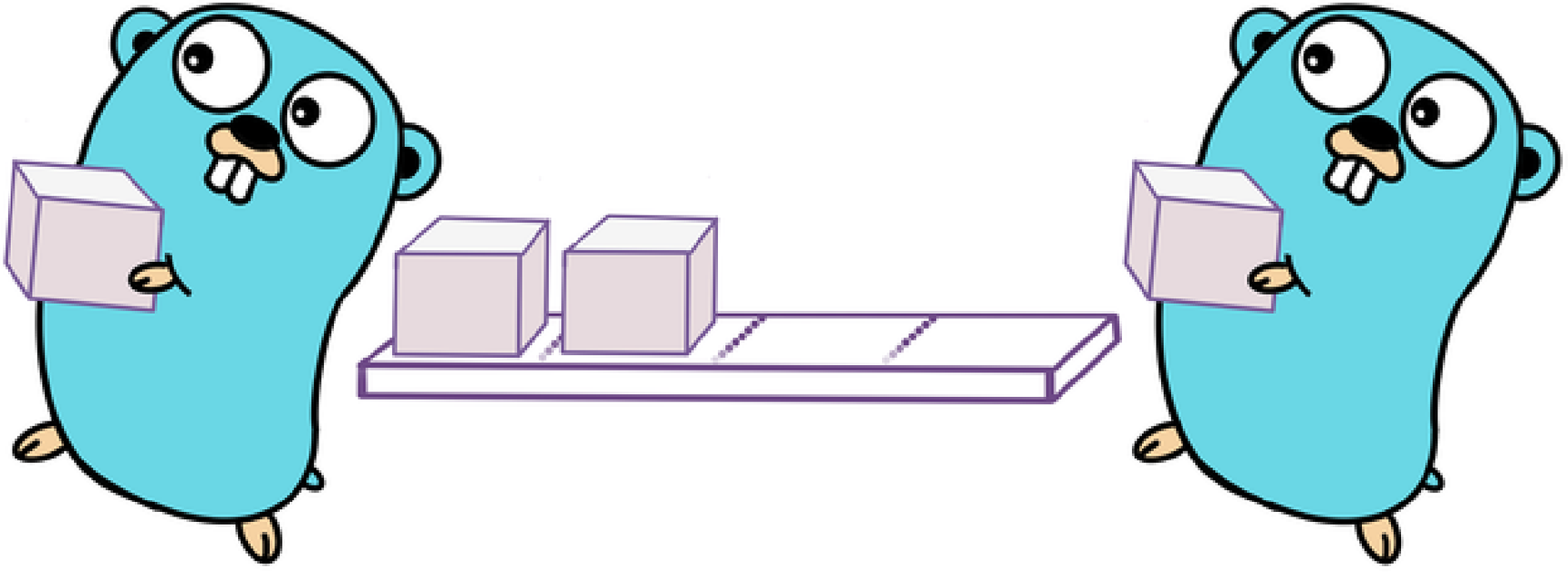
# Каналы: небуферизованные

```
ch := make(chan int)
```



# Каналы: буферизованные

```
ch := make(chan int, 4)
```

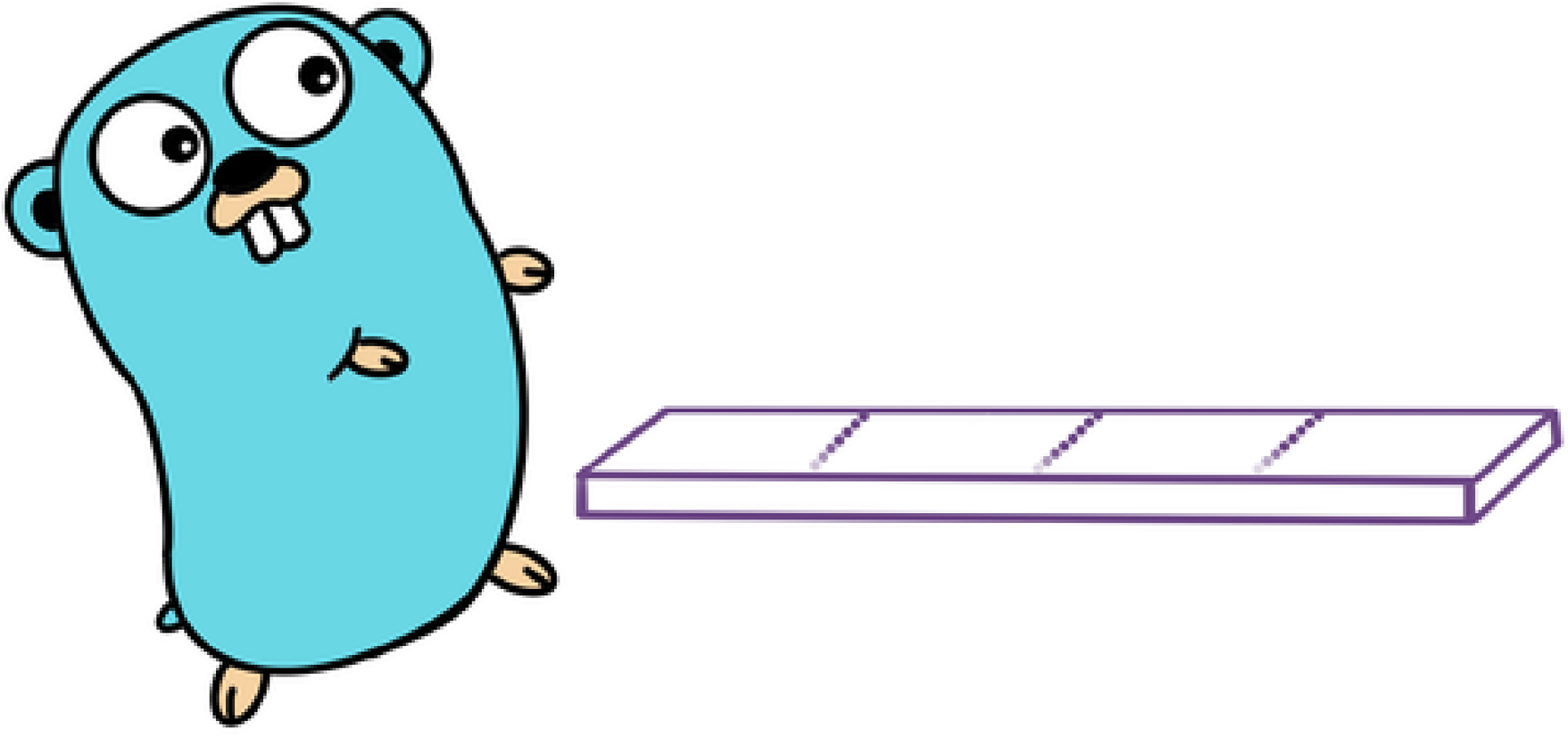


# Каналы: небуферизованные

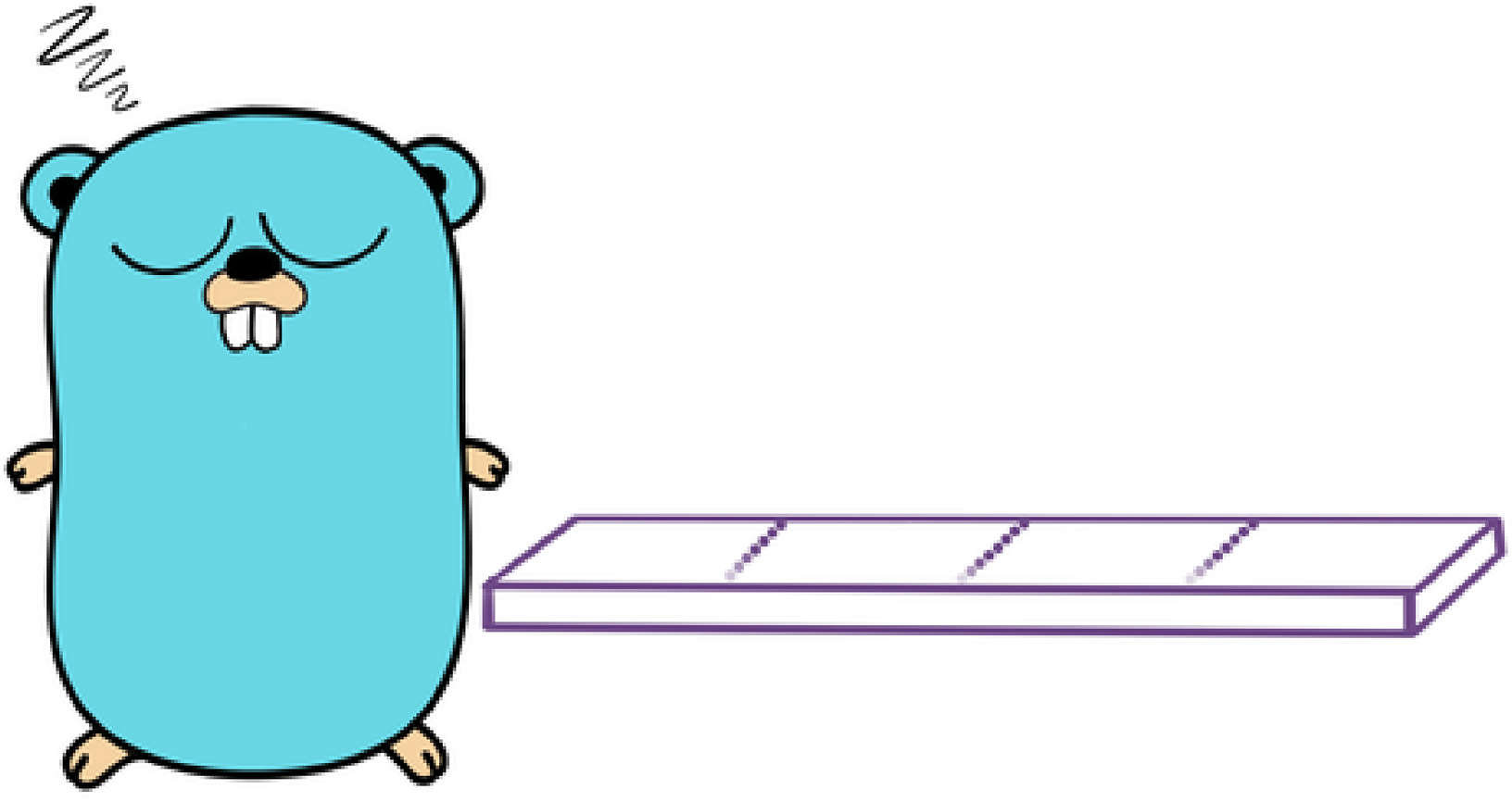
Чему равен буфер небуферизованного канала?

```
ch := make(chan int, ?)
```

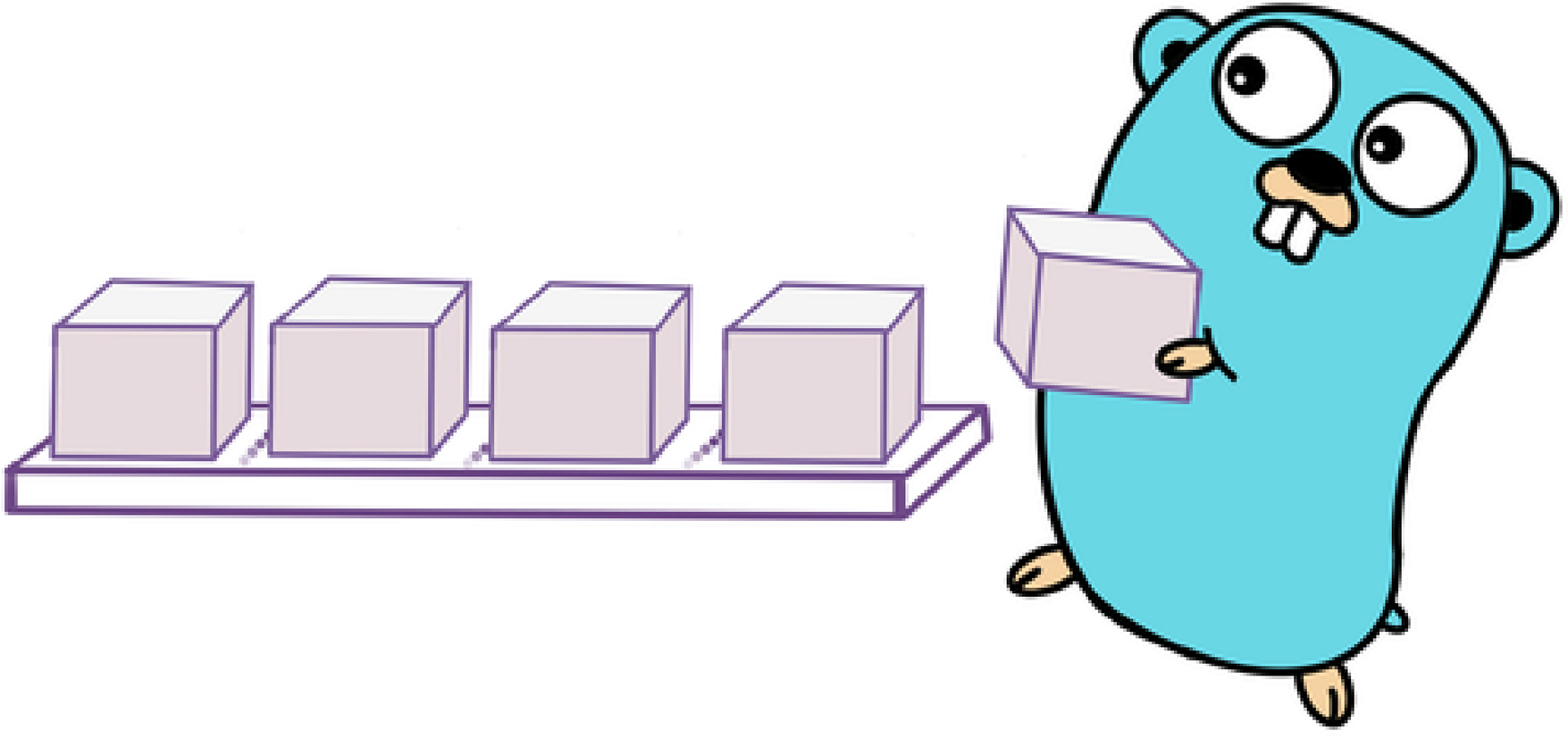
# Что будет, если читать из пустого канала?



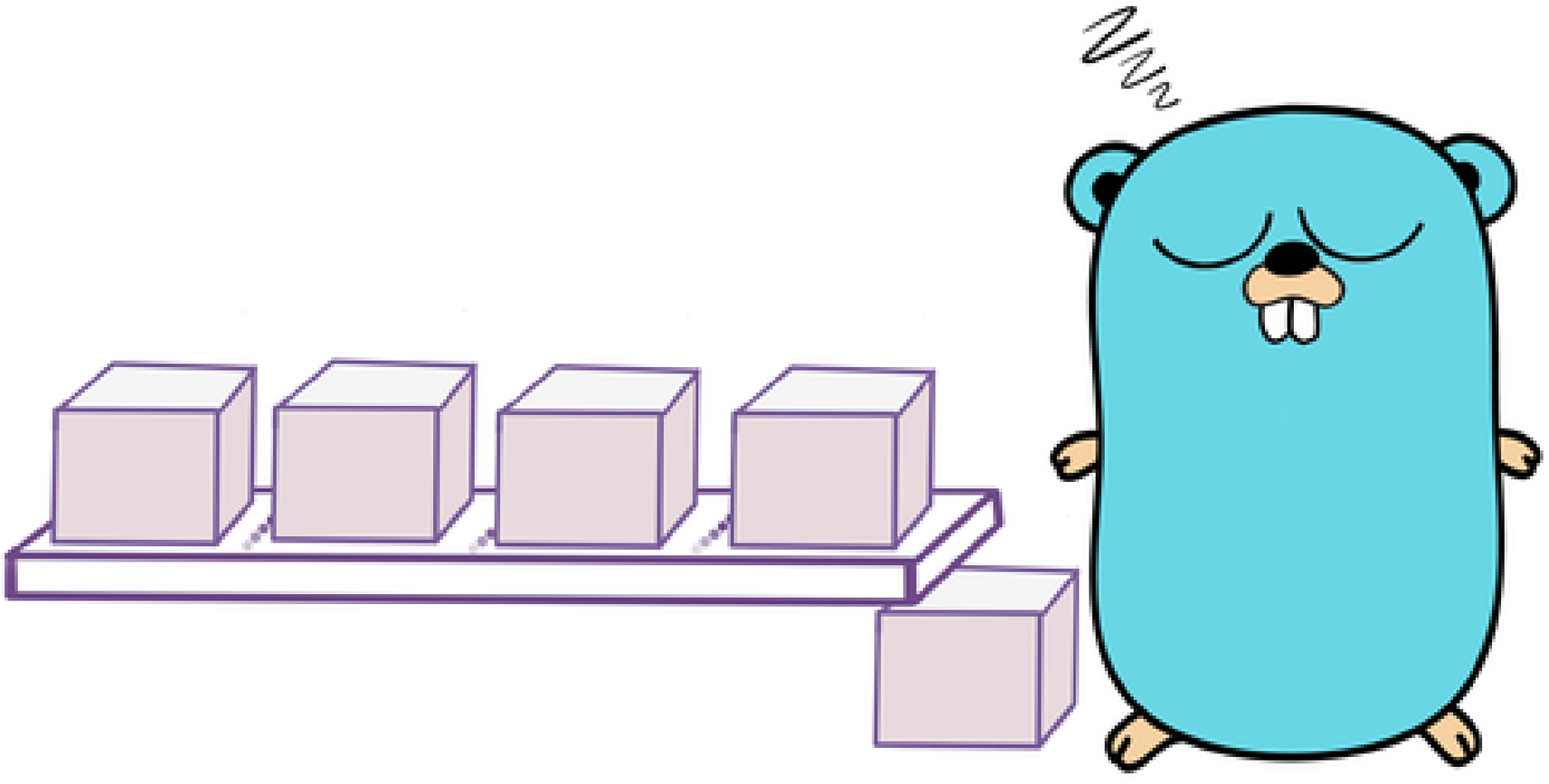
# Что будет, если читать из пустого канала?



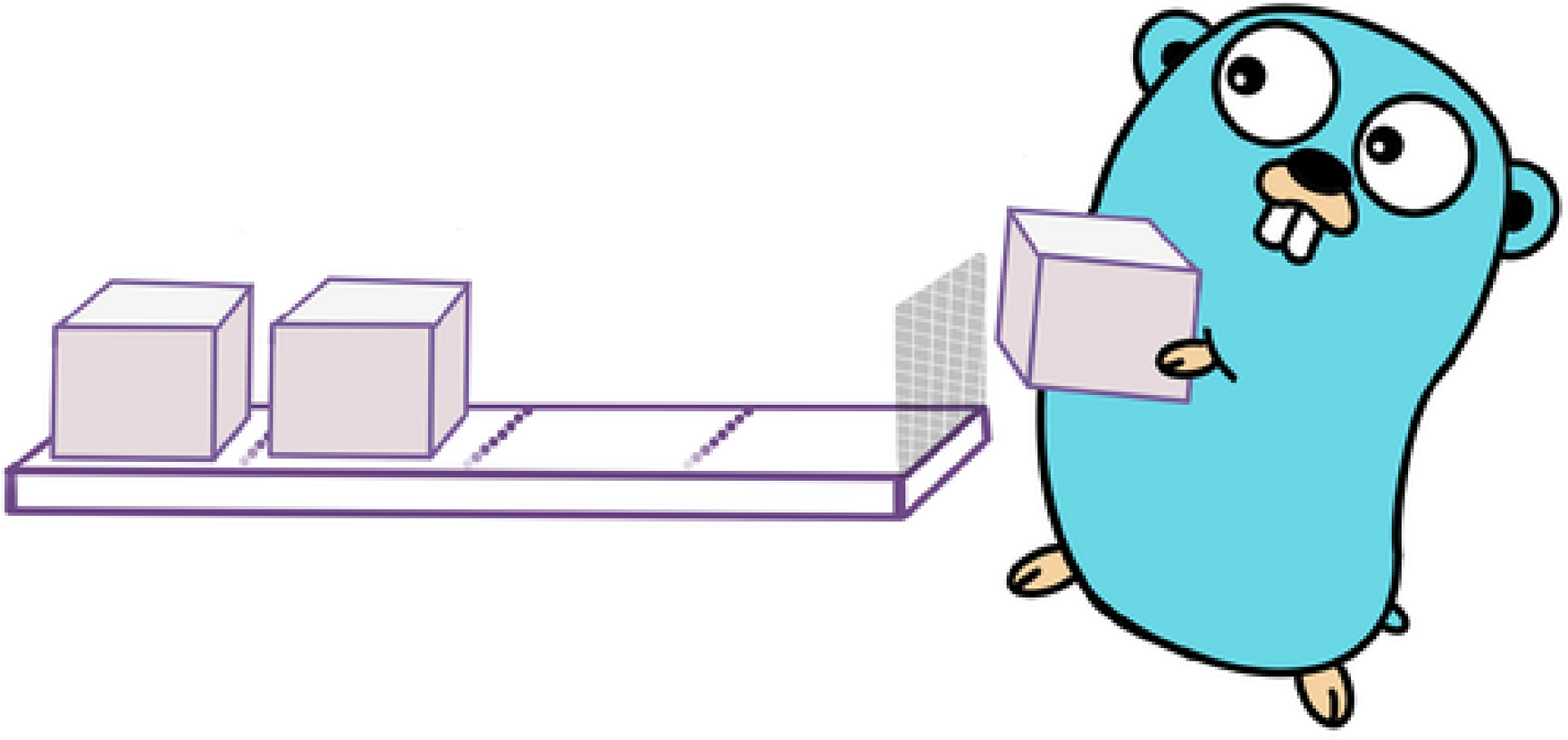
# Что будет, если писать в заполненный канал?



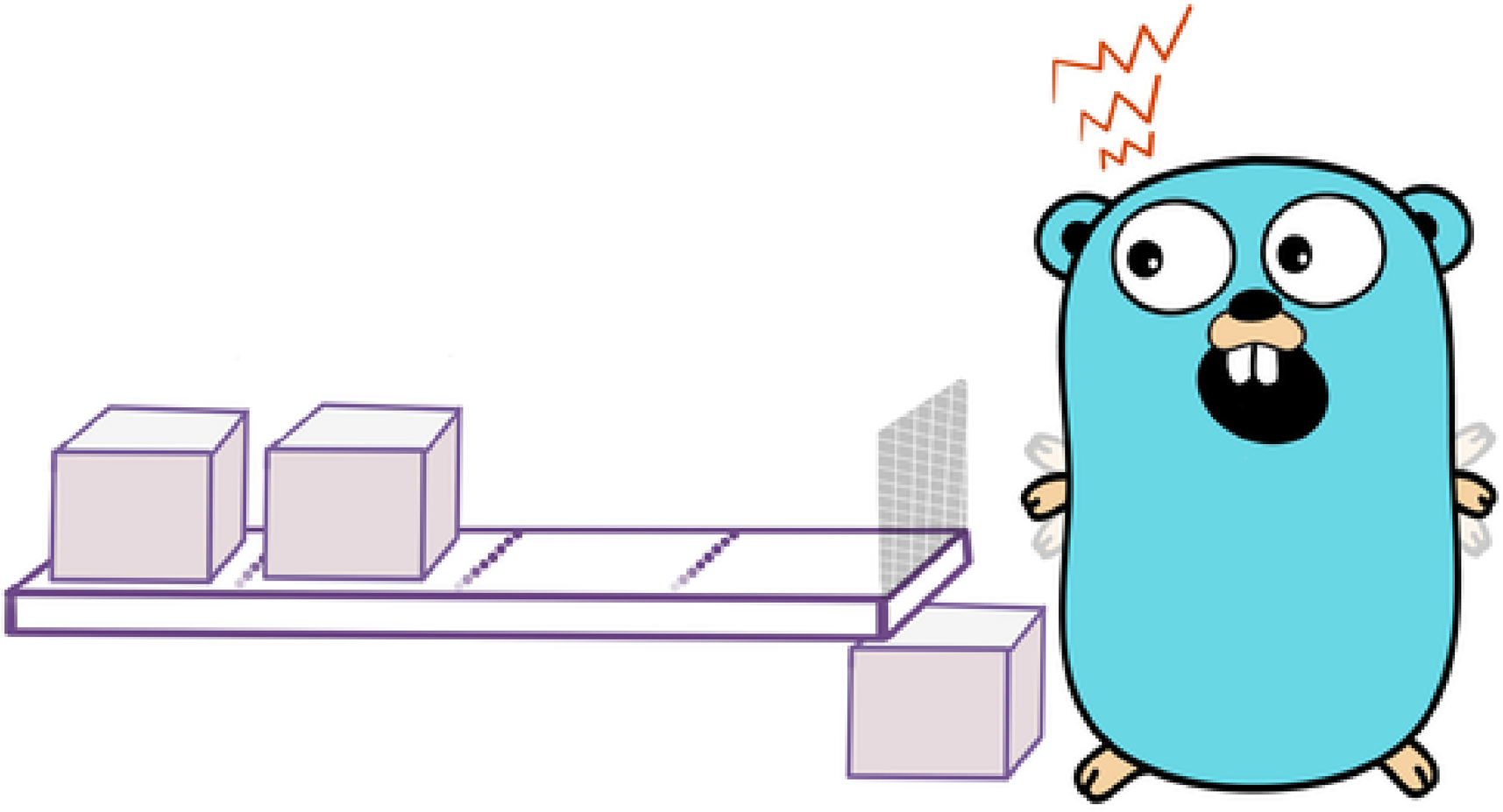
# Что будет, если писать в заполненный канал?



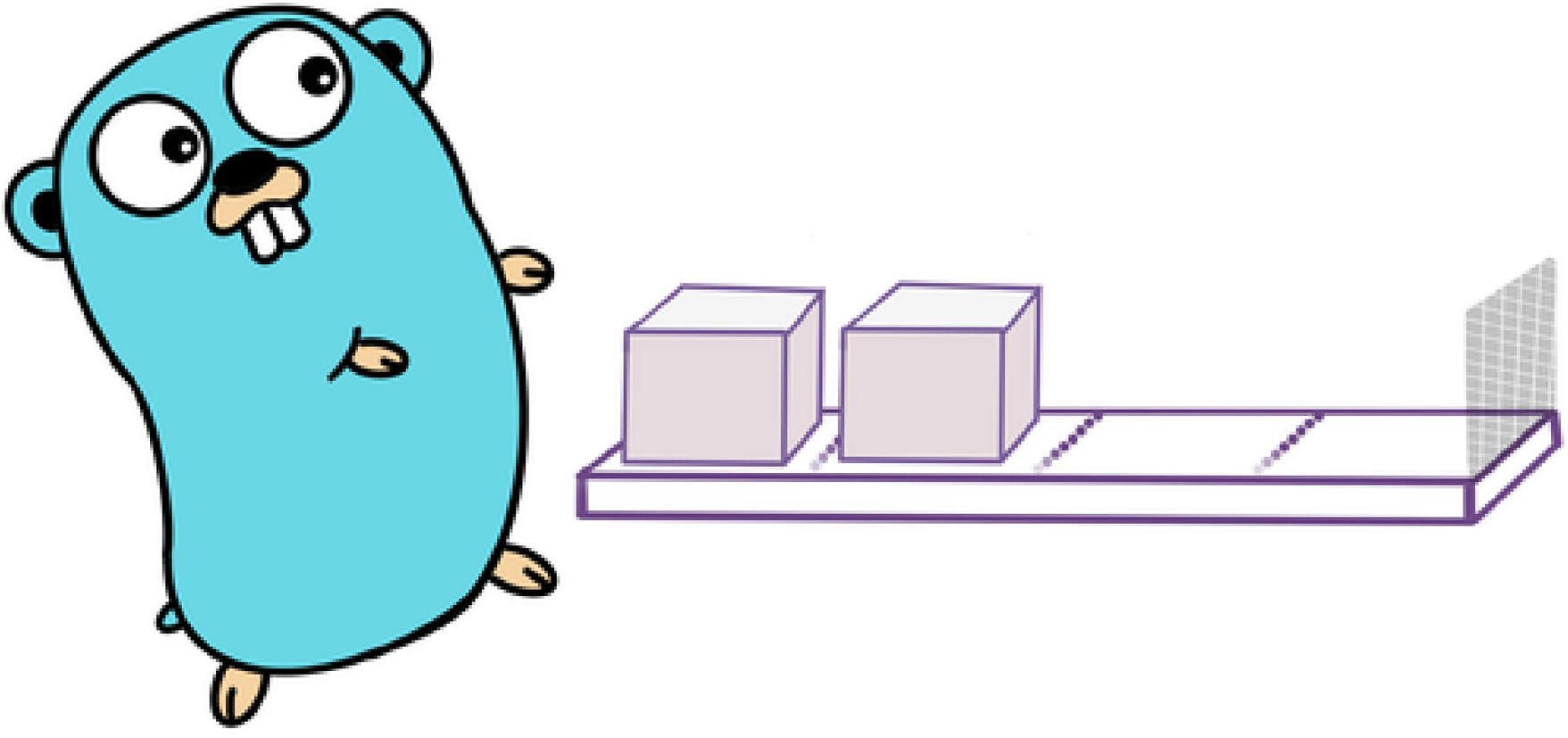
# Что будет, если писать в закрытый канал?



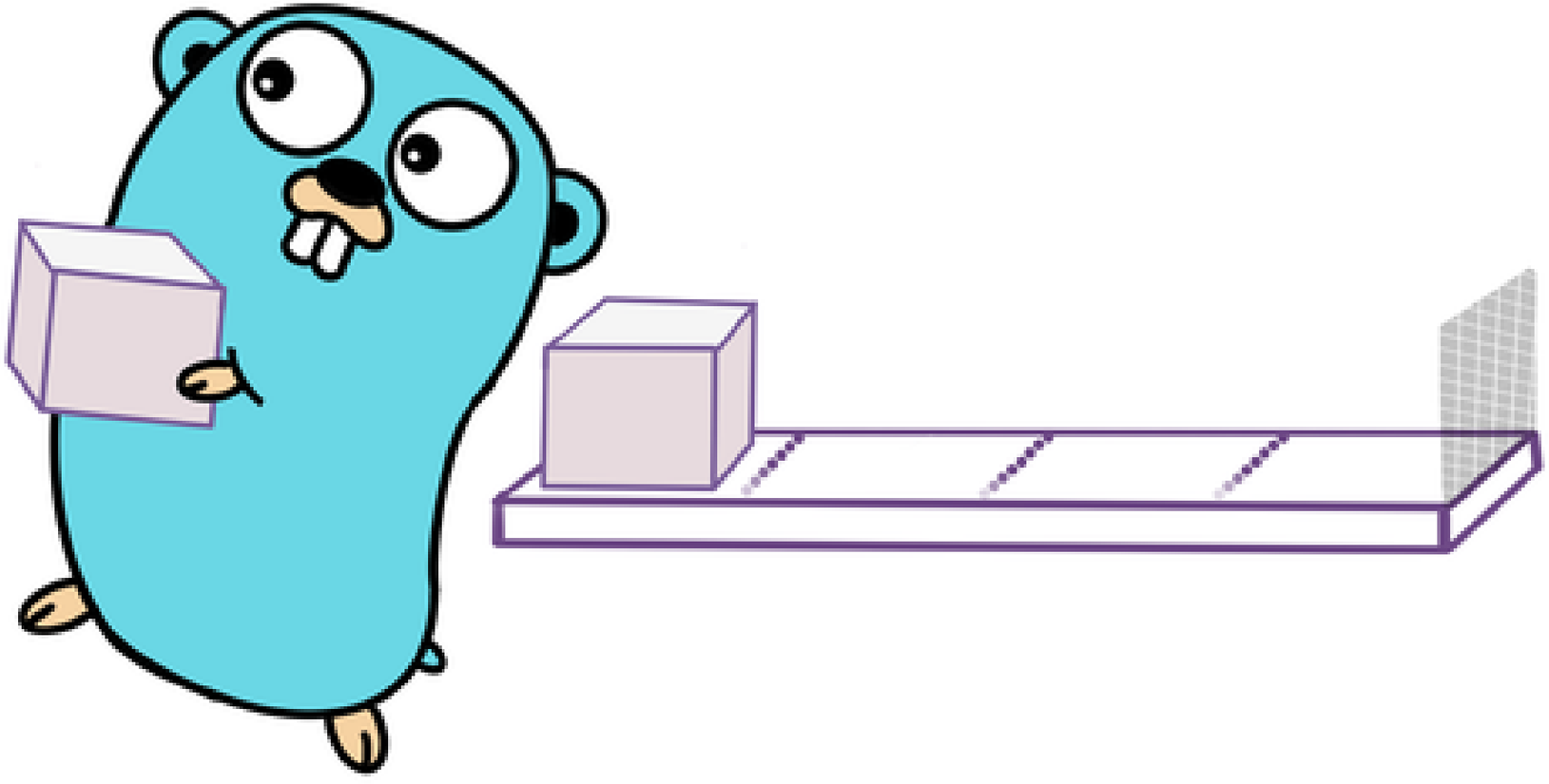
# Что будет, если писать в закрытый канал?



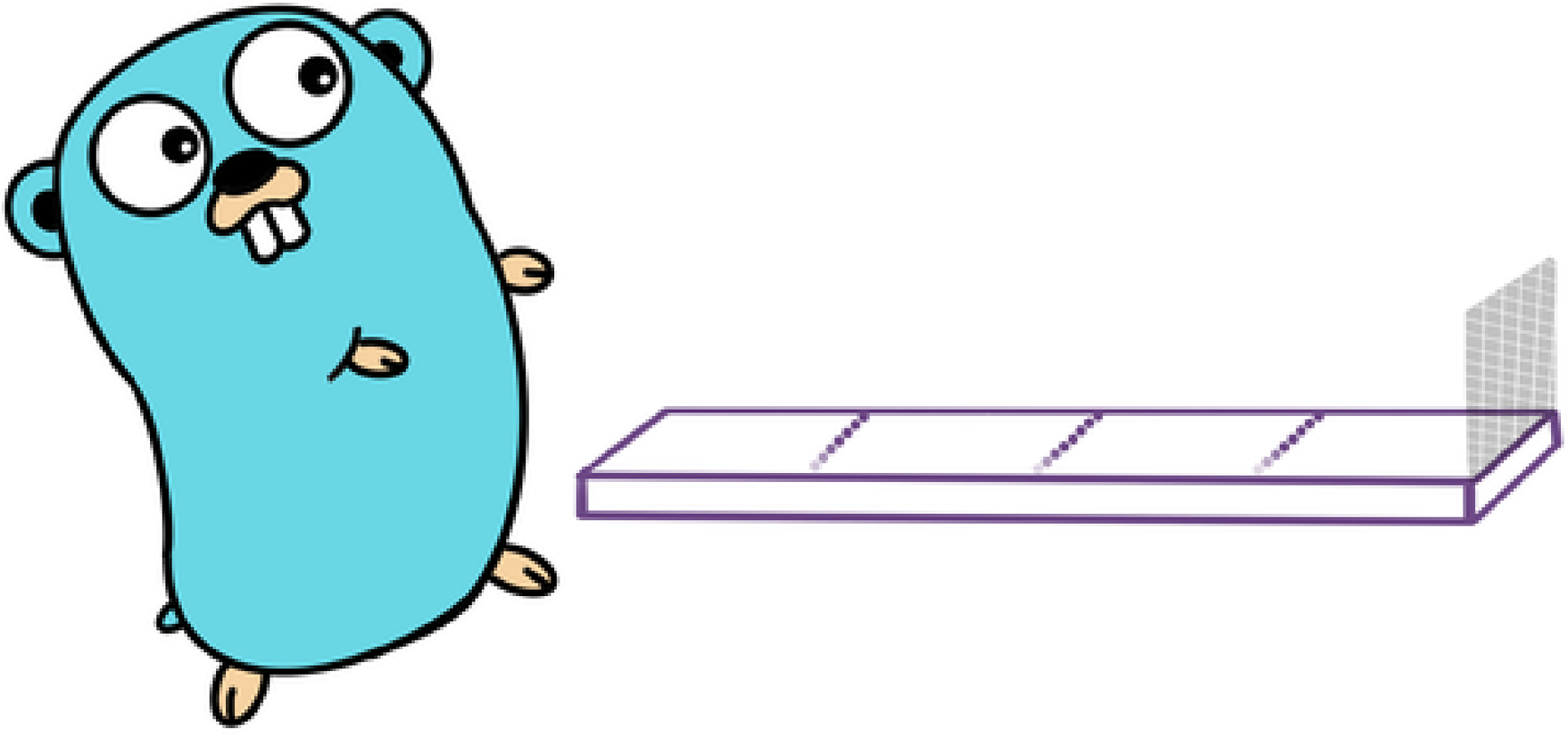
# Что будет, если читать из закрытого канала?



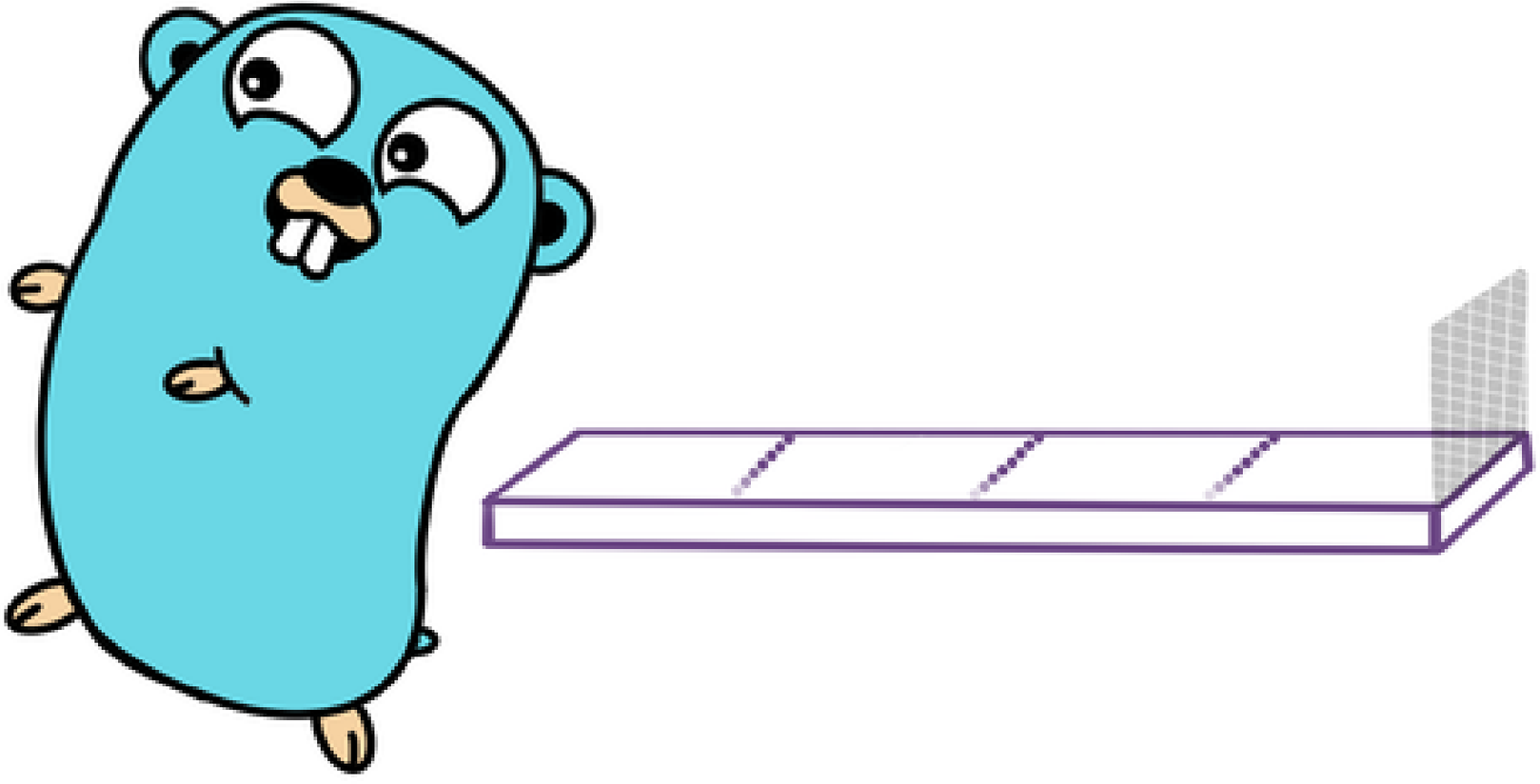
# Что будет, если читать из закрытого канала?



# Что будет, если читать из пустого закрытого канала?



# Что будет, если читать из пустого закрытого канала?



# Синхронизация горутин каналами

```
func main() {
    var ch = make(chan struct{})

    go func() {
        fmt.Printf("Hello")
        ch <- struct{}{}
    }()

    <-ch
}
```

https://goplay.space/#TeLXxeAP0D6

# Синхронизация горутин каналами

```
func main() {
    var ch = make(chan struct{})

    go func() {
        fmt.Printf("Hello")
        <-ch
    }()

    ch <- struct{}{}
}
```

https://goplay.space/#TeLXxeAP0D6

# Чтение из канала, пока он не закрыт

```
v, ok := <-ch // значение и флаг "открытости" канала
```

# Чтение из канала, пока он не закрыт

Producer:

```
for _, t := range tasks {
    ch <- t
}
close(ch)
```

Consumer:

```
for {
    x, ok := <-ch
    if !ok {
        break
    }

    fmt.Println(x)
}
```

# Чтение из канала, пока он не закрыт

Producer:

```
for _, t := range tasks {
    ch <- t
}
close(ch)
```

Consumer:

```
for x := range ch {
    fmt.Println(x)
}
```

# Правила закрытия канала

- Кто закрывает канал?

# Правила закрытия канала

- Канал закрывает тот, кто в него пишет.
- Если несколько писателей, то тот, кто создал писателей и канал.

# Каналы: однонаправленные

```
chan<- T // только запись
<-chan T // только чтение
```

Что произойдет?

```
func f(out chan<- int) {
    <-out
}

func main() {
    var ch = make(chan int)
    f(ch)
}
```

https://goplay.space/#t6bVfgg6BTu

# Каналы: таймаут

```
timer := time.NewTimer(10 * time.Second)
select {
case data := <-ch:
    fmt.Printf("received: %v", data)
case <-timer.C:
    fmt.Printf("failed to receive in 10s")
}
```

https://goplay.space/#40A5bnJQiAk

# Каналы: периодик

```
ticker := time.NewTicker(10 * time.Second)
defer ticker.Stop()

for {
    select {
    case <-ticker.C:
        fmt.Printf("tick")
    case <-doneCh:
        return
    }
}
```

https://goplay.space/#E2wyvzdXYIS

# Каналы: как сигналы

```
make(chan struct{}, 1)
```

Источник сигнала:

```
select {
    case notifyCh <- struct{}{}:
    default:
}
```

Приемник сигнала:

```
select {
    case <-notifyCh:
    case ...
}
```

# Каналы: graceful shutdown

```
interruptCh := make(chan os.Signal, 1)

signal.Notify(interruptCh, os.Interrupt, syscall.SIGTERM)

fmt.Printf("Got %v...\n", <-interruptCh)
```

# sync.WaitGroup: какую проблему решает?

```
func main() {
    const goCount = 5

    ch := make(chan struct{})
    for i := 0; i < goCount; i++ {
        go func() {
            fmt.Println("go-go-go")
            ch <- struct{}{}
        }()
    }

    for i := 0; i < goCount; i++ {
        <-ch
    }
}
```

https://goplay.space/#O0C7h_IsWl8

# sync.WaitGroup: ожидание горутин

```
func main() {
    const goCount = 5

    wg := sync.WaitGroup{}
    wg.Add(goCount) // <===

    for i := 0; i < goCount; i++ {
        go func() {
            fmt.Println("go-go-go")
            wg.Done() // <===
        }()
    }

    wg.Wait()
}
```

https://goplay.space/#u90fGD8vZ_X

# sync.WaitGroup: ожидание горутин

```
func main() {
    wg := sync.WaitGroup{}

    for i := 0; i < 5; i++ {
        wg.Add(1) // <===
        go func() {
            fmt.Println("go-go-go")
            wg.Done() // <===
        }()
    }

    wg.Wait()
}
```

# sync.WaitGroup: API

```
type WaitGroup struct {
}

func (wg *WaitGroup) Add(delta int) - увеличивает счетчик WaitGroup.

func (wg *WaitGroup) Done() - уменьшает счетчик на 1.

func (wg *WaitGroup) Wait() - блокируется, пока счетчик WaitGroup не обнулится.
```

# sync.Once: какую проблему решает?

```
func main() {
    var once sync.Once
    onceBody := func() {
        fmt.Println("Only once")
    }

    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func() {
            once.Do(onceBody)
            wg.Done()
        }()
    }
    wg.Wait()
}
```

https://goplay.space/#VxMyPmXHPzq

# sync.Once: ленивая инициализация (пример)

```
type List struct {
    once sync.Once
    ...
}

func (l *List) PushFront(v interface{}) {
    l.init()
    ...
}

func (l *List) init() {
    l.once.Do(func() {
        ...
    })
}
```

# sync.Once: синглтон (пример)

```
type singleton struct {
}

var instance *singleton
var once sync.Once

func GetInstance() *singleton {
    once.Do(func() {
        instance = &singleton{}
    })
    return instance
}
```

# sync.Mutex: какую проблему решает?

```
func main() {
    wg := sync.WaitGroup{}

    v := 0
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            v++
            wg.Done()
        }()
    }

    wg.Wait()
    fmt.Println(v)
}
```

https://goplay.space/#bf6NKB5z0QO

# sync.Mutex

```
$ GOMAXPROCS=1 go run mu.go
1000
$ GOMAXPROCS=4 go run mu.go
947
$ GOMAXPROCS=4 go run mu.go
956
```

# sync.Mutex

```
func main() {
    wg := sync.WaitGroup{}

    v := 0
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            old_v := v
            new_v := old_v + 1
            v = new_v
            wg.Done()
        }()
    }

    wg.Wait()
    fmt.Println(v)
}
```

# sync.Mutex

Мью́текс (англ. mutex, от mutual exclusion — «взаимное исключение»).

Код между Lock и Unlock выполняет только одна горутина, остальные ждут:

```
mutex.Lock()
v++
mutex.Unlock()
```

# sync.Mutex

```
func main() {
    wg := sync.WaitGroup{}
    mu := sync.Mutex{}

    v := 0
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            mu.Lock()   // <===
            v++
            mu.Unlock() // <===
            wg.Done()
        }()
    }

    wg.Wait()
    fmt.Println(v)
}
```

# sync.Mutex: паттерны использования

Помещайте мьютекс выше тех полей, доступ к которым он будет защищать

```
var sum struct {
    mu sync.Mutex // <=== этот мьютекс защищает
    i  int        // <=== поле под ним
}
```

# sync.Mutex: паттерны использования

Используйте defer, если есть несколько точек выхода

```
func doSomething() {
    mu.Lock()
    defer mu.Unlock()

    err := ...
    if err != nil {
        return // <===
    }

    err = ...
    if err != nil {
        return // <===
    }
    return // <===
}
```

# sync.Mutex: паттерны использования

НО!

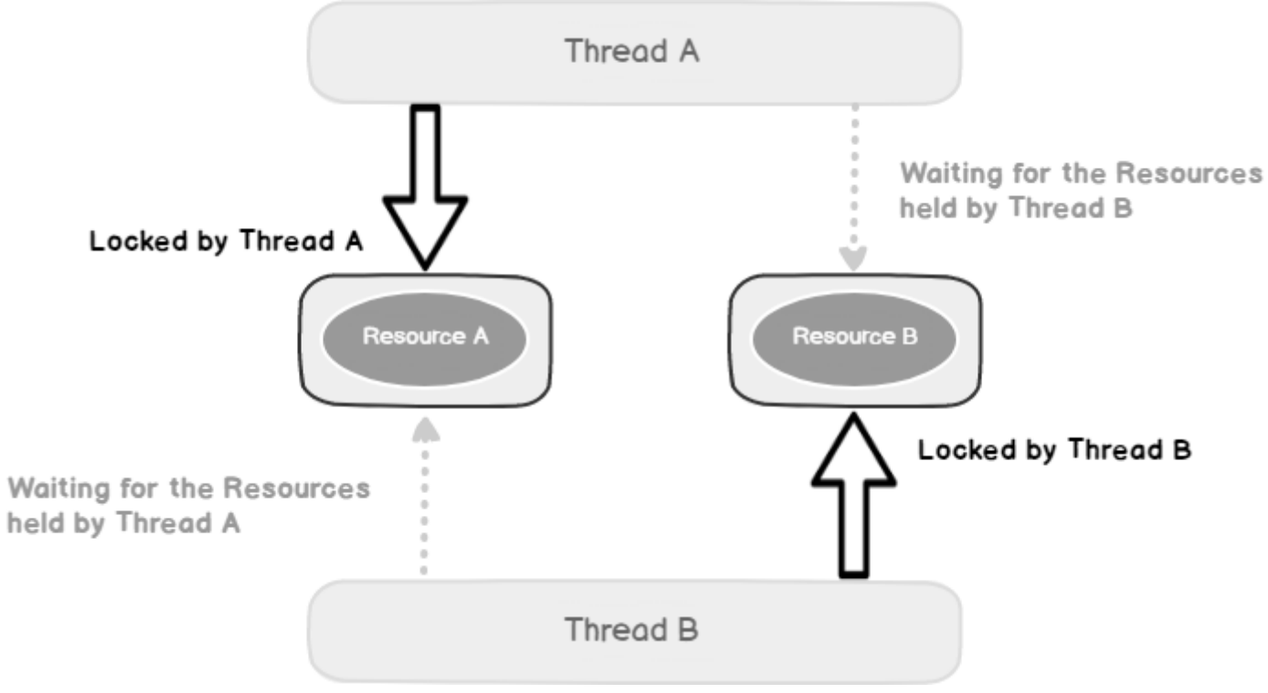
Берегитесь копирования мьютексов

https://goplay.space/#gdGnlc8PiOR

sync.Map

https://www.youtube.com/watch?v=C1EtfDnsdDs

# sync.Mutex: Что такое дедлок?




https://goplay.space/#PLLvZfDiDqs

https://goplay.space/#k37TLDEOu9I

# Race detector

В чем проблема, кроме неопределенного поведения?

```
func main() {
    wg := sync.WaitGroup{}
    text := ""

    wg.Add(2)

    go func() {
        text = "hello world"
        wg.Done()
    }()

    go func() {
        fmt.Println(text)
        wg.Done()
    }()

    wg.Wait()
}
```

# Race detector

```
$ go test -race mypkg
$ go run -race mysrc.go
$ go build -race mycmd
$ go install -race mypkg
```

https://blog.golang.org/race-detector
http://robertknight.github.io/talks/golang-race-detector.html
https://medium.com/german-gorelkin/race-8936927dba20

# Race detector

Ограничение race детектора:

```
func main() {
    for i := 0; i < 10000; i++ {
        go func() {
            time.Sleep(time.Second)
        }()
    }
    time.Sleep(time.Second)
}
```

Можно исключить тесты:

```
// +build !race

package foo

// The test contains a data race. See issue 123.
    func TestFoo(t *testing.T) {
// ...
}
```